prospector

Release 1.1

Benjamin Johnson

Jan 01, 2023

USER GUIDE

Prospector is a package to conduct principled inference of stellar population properties from photometric and/or spectroscopic data using flexible models. Prospector allows you to:

- Infer high-dimensional stellar population properties, including nebular emission, from rest UV through Far-IR data (with nested or ensemble MCMC sampling.)
- Combine photometric and spectroscopic data rigorously using a flexible spectroscopic calibration model and forward modeling many aspects of spectroscopic data analysis.
- Use spectra and/or photometry to constrain highly flexible star formation history treatments.

CHAPTER

ONE

INSTALLATION

Prospector itself is is pure python. To install a released version of just Prospector, use `pip`

```
python -m pip install astro-prospector
```

Then in Python

```
import prospect
print(prospect.__version__)
```

However, several other packages are required for the code to model and fit SEDs (see below.)

1.1 Development Version

To install the development version of Prospector and its dependencies to a conda environment, use the following procedure:

```
# change this if you want to install elsewhere;
# or, copy and run this script in the desired location
CODEDIR=$PWD
cd $CODEDIR

# Clone FSPS to get data files
git clone git@github.com:cconroy20/fsps
export SPS_HOME="$PWD/fsps"

# Create and activate environment (here named 'prospector')
git clone git@github.com:bd-j/prospector.git
cd prospector
conda env create -f environment.yml -n prospector
conda activate prospector
# Install latest development version of prospector
python -m pip uninstall astro-prospector
python -m pip install .

echo "Add 'export SPS_HOME=$SPS_HOME' to your .bashrc"

# To use prospector activate the conda environment
conda activate prospector
```

1.2 Requirements

Prospector works with Python3, and requires numpy and SciPy

You will also need:

- astropy
- emcee and/or dynesty for inference (Please cite these packages in any publications)
- sedpy (for filter projections)
- HDF5 and h5py (If you have Enthought or Anaconda one or both of these may already be installed, or you can get HDF5 from homebrew or macports and h5py via pip)

For modeling galaxies you will need:

- FSPS and python-FSPS (Please cite these packages in any publications)

For parallel processing with emcee or dynesty (optional) you will need:

- MPI (e.g. openMPI or mvapich2, available from homebrew, macports, or Anaconda) and mpi4py

CHAPTER

TWO

USER INTERACTION

The primary user interaction is through a **parameter file**, a python file in which several functions must be defined. These functions are described below and are used to build the ingredients for a fit (data, model, and noise model.) During execution any supplied command line options are parsed – includiing any user defined custom arguments – and the resulting set of arguments is passed to each of these functions before fitting begins.

Command line syntax calls the parameter file and is as follows for single thread execution:

```
python parameter_file.py --dynesty
```

Additional command line options can be given (see below) e.g.

```
python parameter_file.py --emcee --nwalkers=128
```

will cause a fit to be run using emcee with 128 walkers.

A description of the available command line options can be obtained with

```
python parameter_file.py --help
```

All of this command line syntax requires that the end of the parameter file have something like the following code block at the end, which reads command line arguments, runs all the *build_* methods (see below), conducts the fit, and writes output.

```
if __name__ == "__main__":
    import time
    from prospect.fitting import fit_model
    from prospect.io import write_results as writer
            from prospect import prospect_args

    # Get the default argument parser
    parser = prospect_args.get_parser()
    # Add custom arguments that controll the build methods
    parser.add_argument("--custom_argument_1", ...)
    # Parse the supplied arguments, convert to a dictionary, and add this file for␣
↪logging purposes
    args = parser.parse_args()
    run_params = vars(args)
    run_params["param_file"] = __file__

    # build the fit ingredients
    obs, model, sps, noise = build_all(**run_params)
    run_params["sps_libraries"] = sps.ssp.libraries
```

(continues on next page)

(continued from previous page)

```
    # Set up an output file name and run the fit
    ts = time.strftime("%y%b%d-%H.%M", time.localtime())
    hfile = "{0}_{1}_mcmc.h5".format(args.outfile, ts)
    output = fit_model(obs, model, sps, noise, **run_params)

    # Write results to output file
    writer.write_hdf5(hfile, run_params, model, obs,
                      output["sampling"][0], output["optimization"][0],
                      tsample=output["sampling"][1],
                      toptimize=output["optimization"][1],
                      sps=sps)
```

2.1 Command Line Options and Custom Arguments

A number of default command line options are included with prospector. These options can control the output filenames and format and some details of how the model is built and run. However, most of the default parameters control the fitting backends.

You can inspect the default set of arguments and their default values as follows:

```
from prospect import prospect_args
parser = prospect_args.get_parser()
parser.print_help()
```

In the typical **parameter file** the arguments are converted to a dictionary and passed as keyword arguments to all of the `build_*()` methods described below.

A user can add custom arguments that will further control the behavior of the model and data building methods. This is done by adding arguments to the parser in the executable part of the **parameter file**. See the argparse documentation for details on adding custom arguments.

2.2 Build methods

The required methods in a **parameter file** for building the data and model are:

1. `build_obs()`: This function will take the command line arguments dictionary as keyword arguments and returns on obs dictionary (see *Data Formats* .)
2. `build_model()`: This function will take the command line arguments dictionary dictionary as keyword arguments and return an instance of a `ProspectorParams` subclass, containing information about the parameters of the model (see *Models* .)
3. `build_sps()`: This function will take the command line arguments dictionary dictionary as keyword arguments and return an **sps** object, which must have the method `get_spectrum()` defined. This object generally includes all the spectral libraries and isochrones necessary to build a model, as well as much of the model building code and as such has a large memory footprint.
4. `build_noise()`: This function should return a `NoiseModel` object for the spectroscopy and/or photometry. Either or both can be `None` (the default) in which case the likelihood will not include covariant noise or jitter and is equivalent to basic χ^2.

2.3 Using MPI

For large galaxy samples we recommend conducting a fit for each object entirely independently on individual CPU cores. However, for a small number of objects or during testing it can be helpful to decrease the elapsed wall time for a single fit. Message Passing Interface (MPI) can be used to parallelize the fit for a single object over many CPU cores. This will reduce the wall time required for a single fit, but will not reduce the total CPU uptime (and when using dynesty might actually increase the total CPU usage).

To use MPI a "pool" of cores must be made available; each core will instantiate the fitting ingredients separately, and a single core in the pool will then conduct the fit, distributing likelihood requests to the other cores in the pool. This requires changes to the final code block that instantiates and runs the fit:

```
if __name__ == "__main__":
    import time
    from prospect.fitting import fit_model
    from prospect.io import write_results as writer
            from prospect import prospect_args

    # Get the default argument parser
    parser = prospect_args.get_parser()
    # Add custom arguments that controll the build methods
    parser.add_argument("--custom_argument_1", ...)
    # Parse the supplied arguments, convert to a dictionary, and add this file for␣
↪logging purposes
    args = parser.parse_args()
    run_params = vars(args)
    run_params["param_file"] = __file__

    # Build the fit ingredients on each process
    obs, model, sps, noise = build_all(**run_params)
    run_params["sps_libraries"] = sps.ssp.libraries

    # Set up MPI communication
    try:
        import mpi4py
        from mpi4py import MPI
        from schwimmbad import MPIPool

        mpi4py.rc.threads = False
        mpi4py.rc.recv_mprobe = False

        comm = MPI.COMM_WORLD
        size = comm.Get_size()

        withmpi = comm.Get_size() > 1
    except ImportError:
        print('Failed to start MPI; are mpi4py and schwimmbad installed? Proceeding␣
↪without MPI.')
        withmpi = False

# Evaluate SPS over logzsol grid in order to get necessary data in cache/memory
# for each MPI process. Otherwise, you risk creating a lag between the MPI tasks
# caching data depending which can slow down the parallelization
```

(continues on next page)

(continued from previous page)

```
if (withmpi) & ('logzsol' in model.free_params):
    dummy_obs = dict(filters=None, wavelength=None)

    logzsol_prior = model.config_dict["logzsol"]['prior']
    lo, hi = logzsol_prior.range
    logzsol_grid = np.around(np.arange(lo, hi, step=0.1), decimals=2)

    sps.update(**model.params)  # make sure we are caching the correct IMF / SFH / etc
    for logzsol in logzsol_grid:
        model.params["logzsol"] = np.array([logzsol])
        _ = model.predict(model.theta, obs=dummy_obs, sps=sps)

# ensure that each processor runs its own version of FSPS
# this ensures no cross-over memory usage
from prospect.fitting import lnprobfn
from functools import partial
lnprobfn_fixed = partial(lnprobfn, sps=sps)

if withmpi:
    run_params["using_mpi"] = True
    with MPIPool() as pool:

        # The dependent processes will run up to this point in the code
        if not pool.is_master():
            pool.wait()
            sys.exit(0)
        nprocs = pool.size
        # The parent process will oversee the fitting
        output = fit_model(obs, model, sps, noise, pool=pool, queue_size=nprocs,␣
↪lnprobfn=lnprobfn_fixed, **run_params)
else:
    # without MPI we don't pass the pool
    output = fit_model(obs, model, sps, noise, lnprobfn=lnprobfn_fixed, **run_params)

# Set up an output file and write
ts = time.strftime("%y%b%d-%H.%M", time.localtime())
hfile = "{0}_{1}_mcmc.h5".format(args.outfile, ts)
writer.write_hdf5(hfile, run_params, model, obs,
                  output["sampling"][0], output["optimization"][0],
                  tsample=output["sampling"][1],
                  toptimize=output["optimization"][1],
                  sps=sps)

try:
    hfile.close()
except(AttributeError):
    pass
```

Then, to run this file using mpi it can be called from the command line with something like

```
mpirun -np <number of processors> python parameter_file.py --emcee
# or
```

(continues on next page)

(continued from previous page)

```
mpirun -np <number of processors> python parameter_file.py --dynesty
```

Note that only model evaluation is parallelizable with *dynesty*, and many operations (e.g. new point proposal) are still done in serial. This means that single-core fits will always be more efficient in terms of total CPU usage per fit. Having a large ratio of (live points / processors) helps efficiency, the scaling goes as K ln(1 + M/K), where M = number of processes and K = number of live points.

For *emcee* efficiency is maximized when K/(M-1) is an integer >= 2, where M = number of processes and K = number of walkers. The wall time speedup should be approximately the same as this integer.

CHAPTER

THREE

DATA FORMATS

3.1 The obs Dictionary & Data Units

Prospector expects the data in the form of a dictionary, preferably returned by a `build_obs()` function (see below). This dictionary should have (at least) the following keys and values:

"wavelength" The wavelength vector for the spectrum, ndarray. Units are vacuum Angstroms. The model spectrum will be computed for each element of this vector. Set to `None` if you have no spectrum. If fitting observed frame photometry as well, then these should be observed frame wavelengths.

"spectrum" The flux vector for the spectrum, ndarray of same length as the wavelength vector. If absolute spectrophotometry is available, the units of this spectrum should be Janskies divided by 3631 (i.e. maggies). Also the `rescale_spectrum` run parameter should be False.

"unc" The uncertainty vector (sigma), in same units as `"spectrum"`, ndarray of same length as the wavelength vector.

"mask" A boolean array of same length as the wavelength vector, where `False` elements are ignored in the likelihood calculation.

"filters" A sequence of sedpy filter objects or filter names, used to calculate model magnitudes.

"maggies" An array of photometric flux densities, same length as `"filters"`. The units are *maggies*. Maggies are a linear flux density unit defined as $\text{maggie} = 10^{-0.4\, m_{AB}}$ where m_{AB} is the AB apparent magnitude. That is, 1 maggie is the flux density in Janskys divided by 3631. Set to `None` if you have no photometric data.

"maggies_unc" An array of photometric flux uncertainties, same length as `"filters"`, that gives the photometric uncertainties in units of *maggies*

"phot_mask" Like `"mask"`, a boolean array, used to mask the photometric data during the likelihood calculation. Elements with `False` values are ignored in the likelihood calculation.

If you do not have spectral or photometric data, you can set `"wavelength": None` or `"maggies": None` respectively. Feel free to add keys that store other metadata, these will be stored on output. However, for ease of storage these keys should either be numpy arrays or basic python datatypes that are JSON serializable (e.g. strings, ints, and floats and lists, dicts, and tuples thereof.)

The method `prospect.utils.obsutils.fix_obs()` can be used as a shortcut to add any of the missing required keys with their default values and ensure that there is data to fit, e.g.

```
from prospect.utils.obsutils import fix_obs
# dummy observation dictionary with just a spectrum
N = 1000
obs = dict(wavelength=np.linspace(3000, 5000, N), spectrum=np.zeros(N), unc=np.ones(N))
obs = fix_obs(obs)
assert "mask" in obs.keys()
```

It is recommended to use this method at the end of any *build_obs* function.

3.2 The `build_obs()` function

The `build_obs()` function in the parameter file is written by the user. It should take a dictionary of command line arguments as keyword arguments. It should return an `obs` dictionary described above.

Other than that, the contents can be anything. Within this function you might open and read FITS files, ascii tables, HDF5 files, or query SQL databases. You could, using e.g. an `objid` parameter, dynamically load data (including filter sets) for different objects in a table. Feel free to import helper functions, modules, and packages (like astropy, h5py, sqlite, astroquery, etc.)

The point of this function is that you don't have to *externally* convert your data format to be what Prospector expects and keep another version of files lying around: the conversion happens *within* the code itself. Again, the only requirement is that the function can take a `run_params` dictionary as keyword arguments and that it return an `obs` dictionary as described below.

CHAPTER

FOUR

MODELS

The modular nature of Prospector allows it to be applied to a variety of data types and different scientific questions. However, this flexibility requires the user to take care in defining the model, including prior distributions, which may be specific to a particular scientific question or to the data to be fit. Different models and prior beliefs may be more or less appropriate for different kinds of galaxies. Certain kinds of data may require particular model components in order to be described well. Prospector strives to make this model building process as straightforward as possible, but it is a step that cannot be skipped.

The choice of which parameters to include, which to let vary, and what prior distributions to use will depend on the data being fit (including the types of objects) and the goal of the inference. As examples, if no infrared data is available then it is not necessary to fit – or even include in the model at all – the dust emission parameters controlling the shape of the infrared SED. For globular clusters or completely quenched galaxies it may not be necessary to include nebular emission in the model, and one may wish to adjust the priors on population age or star formation history to be more appropriate for such objects. If spectroscopic data is being fit then it may be necessary to include velocity dispersion as a free parameter. Generating and fitting mock data can be an extremely useful tool for exploring the sensitivity of a given type of data to various parameters.

4.1 Parameter Specification

A model is defined by a dictionary of parameter specifications, keyed by parameter name, that is used to instantiate and configure the model objects (instances of `models.ProspectorParams` or its subclasses.) This dictionary is usually constructed or given in a **parameter file**.

For a single parameter the specification is a dictionary that should at minimum include several keys:

`"N"` An integer specifying the length of the parameter. If not supplied this defaults to `1`, the common case of a scalar parameter.

`"isfree"` Boolean specifying whether a parameter is free to vary during optimization and sampling (`True`) or not (`False`). This defaults to `False` if not supplied.

`"init"` The initial value of the parameter. If the parameter is not free, then this is the value that will be used throughout optimization and sampling. If the parameter is free to vary, this is where optimization will start from or – if no optimization happens – this will be the center of the initial ball of *emcee* walkers. Note that if using nested sampling then the value of `"init"` is not important (though a value must still be given).

For parameters with `isfree=True` the following additional key is required:

`"prior"` An instance of a prior object, including parameters for the prior (e.g. `priors.TopHat(mini=10, maxi=12)`).

If using `emcee`, the following key can be useful to have:

`"init_disp"` The dispersion in this parameter to use when generating an `emcee` sampler ball. This is not technically required, as it defaults to 10% of the initial value. It is ignored if nested sampling is used.

It's also a good idea to have a "units" key, a string describing the units of the the parameter. So, in the end, this looks something like:

```
mass = dict(N=1, init=1e9, isfree=True,
            prior= priors.LogUniform(mini=1e7, maxi=1e12),
            units="M$_\odot$ of stars formed.", init_disp=1e8)
model_params = dict(mass=mass)
```

Nearly all parameters used by FSPS can become a model parameter. When fitting galaxies the default python-FSPS parameter values will be used unless specified in a fixed parameter, e.g. `imf_type` can be changed by including it as a fixed parameter with value given by "`init`".

Parameters can also be used to control the Prospector-specific parts of the modeling code. These include things like spectral smoothing, wavelength calibration, spectrophotometric calibration, and any parameters of the noise model. Be warned though, if you include a parameter that does not affect the model the code will not complain, and if that parameter is free it will simply result in a posterior PDF that is the same as the prior (though optimization algorithms may fail).

4.2 Priors

All parameters that are free to vary must have an associated prior distribution. Prior objects can be found in the *prospect.models.priors* module. When specifying a prior using an object, you can and should specify the parameters of that prior on initialization, e.g.

```
model_params["dust2"]["prior"] = priors.ClippedNormal(mean=0.3, sigma=0.5, mini=0.0,␣
↪maxi=3.0)
```

4.3 Transformations

Sometimes the native parameterization of stellar population models is not the most useful. In these cases parameter *transformations* can prove useful.

Transformations are useful to impose parameter limits that are a function of other parameters; for example, when fitting for redshift it can be useful to reparameterize the age of a population (say, in Gyr) into its age as a fraction of the age of the universe at that redshift. This avoids the problem of populations that are older than the age of the universe, or complicated joint priors on the population age and the redshift. A number of useful transformation functions are provided in Prospector and these may be easily supplemented with user defined functions.

This parameter transformation and dependency mechanism can be used to tie any number of parameters to any number of other parameters in the model, as long as the latter parameters are not *also* dependent on some parameter transformation. This mechanism may also be used to avoid joint priors. For example, if one wishes to place a prior on the ratio of two parameters (say, that it be less than one) then the ratio itself can be introduced as a new parameter, and one of the original parameters can be "fixed" but have its value at each parameter location depend on the other original parameter and the new ratio parameter.

As a simple example, we consider sampling in the log of the SF timescale instead of the timescale itself. The follwing code

```
def delogify(logtau=0, **extras):
    return 10**logtau

model_params["tau"]["isfree"] = False
```

(continues on next page)

(continued from previous page)

```
model_params["tau"]["depends_on"] = delogify
model_params["logtau"] = dict(N=1, init=0, isfree=True, prior=priors.TopHat(mini=-1,␣
↪maxi=1))
```

could be used to set the value of `tau` using the free parameter `logtau` (i.e., sample in the log of a parameter, though setting a `prospect.models.priors.LogUniform` prior is equivalent in terms of the posterior).

This dependency function must take optional extra keywords (`**extras`) because the entire parameter dictionary will be passed to it. Then add the new parameter specification to the `model_params` dictionary for the parameter that can vary (and upon which the fixed parameter depends). In this example that new free parameter would be `logtau`.

This pattern can also be used to tie arbitrary parameters together (e.g. gas-phase and stellar metallicity) while still allowing them to vary. A parameter may depend on multiple other (free or fixed) parameters, and multiple parameters may depend on a single other (free or fixed) parameter. This mechanism is used extensively for the non-parametric SFHs, and is recommended for complex dust attenuation models.

Note: It is important that any parameter with the `"depends_on"` key present is a fixed parameter. For portability and easy reconstruction of the model it is important that the `depends_on` function either be importable (e.g. one of the functions supplied in *prospect.models.transforms*) or defined within the parameter file.

4.4 Parameter Set Templates

A number of predefined sets of parameters (with priors) are available as dictionaries of model specifications from `prospect.models.templates.TemplateLibrary`, these can be a good starting place for building your model. To see the available parameter sets to inspect the free and fixed parameters in a given set, you can do something like

```
        from prospect.models.templates import TemplateLibrary
        # Show all pre-defined parameter sets
        TemplateLibrary.show_contents()
        # Show details on the "parameteric" set of parameters
        TemplateLibrary.describe("parametric_sfh")
        # Simply print all parameter specifications in "parametric_sfh"
        print(TemplateLibrary["parametric_sfh"])
        # Actually get a copy of one of the predefined sets
        model_params = TemplateLibrary["parametric_sfh"]
# This dictionary can be updated or modified, to expand the model.
model_params.update(TemplateLibrary["nebular"])
        # Instantiate a model object
        from prospect.models import SedModel
        model = SedModel(model_params)
```

4.5 The `build_model()` Method

This method in the **parameter file** should take the `run_params` dictionary as keyword arguments, and return an instance of a subclass of `prospect.models.ProspectorParams`.

The model object, a subclass of `prospect.models.ProspectorParams`, is initialized with a list or dictionary (keyed by parameter name) of each of the model parameter specifications described above. If using a list, the order of the list sets the order of the free parameters in the parameter vector. The free parameters will be varied by the code during the optimization and sampling phases. The initial value from which optimization is begun is set by the `"init"` values of each parameter. For fixed parameters the `"init"` value gives the value of that parameter to use throughout the optimization and sampling phases (unless the `"depends_on"` key is present, see advanced.)

The `run_params` dictionary of arguments (including command line modifications) can be used to change how the model parameters are specified within this method before the `prospect.models.ProspectorParams` model object is instantiated. For example, the value of a fixed parameter like `zred` can be set based on values in `run_params` or additional parameters related to dust or nebular emission can be optionally added based on switches in `run_params`.

Useful model objects include `prospect.models.SpecModel` and `prospect.models.PolySpecModel`. The latter includes tools for optimization of spectrophotometric calibration.

CHAPTER

FIVE

SFH TREATMENTS

Numerous star formation history (SFH) treatments are available in prospector. Some of these are described below, along with instructions for their use.

5.1 SSPs

Simple or single stellar populations (SSPs) describe the spectra and properties of a group of stars (withi initial mass distribution described by the IMF) with the same age and metallicity. That is, the SFh is a delta-function in both time in metallicity.

Use of SSP SFHs requires an instance of *prospect.sources.CSPSpecBasis* to be used as the `sps` object. A set of propsector parameters implementing this treatment is available as the `"ssp"` entry of `prospect.models.templates.TemplateLibrary`.

5.2 Parametric SFH

So called "parametric" SFHs describe the SFR as a function of time via a relatively simple function with just a few parameters. In prospector the parametric SFH treatment is actually handled by FSPS itself, and so the model parameters required are the same as those in FSPS (see documentation).

The available parametric SFHs include exponential decay ("tau" models, $\mathrm{SFR} \sim e^{-\mathrm{tage}/\mathrm{tau}}$), and delayed exponential ("delayed-tau" models, $\mathrm{SFR} \sim \mathrm{tage}\, e^{-\mathrm{tage}/\mathrm{tau}}$). To these it is possible to add a burst and/or a truncation, and a constant component can also be added. Finally, the SFH descibed in simha14 is also available. See the FSPS documentation for details.

It is also possible to model *linear combinations* of these parameteric SFHs. This is accomplished by making the `mass` parameter a vector with the number of elements corresponding to the number of components. Other paramaters of the FSPS stellar population model (e.g. `age`, `tau`, and even `dust2` or `dust1`) can be also be made vectors, with vector priors if they are free to be fit; relevant scalar parameters will be shared by all components.

Use of parametric SFHs requires an instance of *prospect.sources.CSPSpecBasis* to be used as the `sps` object. A set of propsector parameters implementing this treatment (defaulting to a delay-tau form, `sfh=4`) is available as the `"parametric_sfh"` entry of `prospect.models.templates.TemplateLibrary`.

5.3 Binned SFHs

The binned or "non-parametric" SFHS are a more flexible alternative to the "parametric" SFHs described above. Rather than being "non-parametric" they actually rely on various parameterizations that fundamentally describe a piece-wise constant SFH, where the SFR is constant within each of a user defined set of temporal bins.

Use of these piece-wise constant SFHs requires an instance of *prospect.sources.FastStepBasis* to be used as the `sps` object. Fundamentally this class requires two vector parameters to generate a model:

- `agebins` an array of shape `(Nbin, 2)` describing the lower and upper *lookback time* of each bin (in units of log(years))
- `mass` an array of shape `(Nbin,)` describing the total stellar mass **formed** in each bin. For the ith bin this means $\mathrm{SFR}_i = \mathrm{mass}_i/(10^{\mathrm{agebins}_{i,1}} - 10^{\mathrm{agebins}_{i,0}})$

The SFH treatments described below all differ in how they transform from the sampled SFH parameters to these fundamental binned SFH parameters, and in the priors placed on those sampled parameters. The transformations between the sampling parameters and these fundamental parameters are given by methods within *prospect.models.transforms*

5.3.1 Continuity SFH

See leja19, johnson21 for more details. A set of propsector parameters implementing this treatment with 3 bins is available as the `"continuity_sfh"` entry of `prospect.models.templates.TemplateLibrary`.

In this parameterization, the SFR of each bin is derived from sampling a vector of parameters describing the *ratio* of SFRs in adjacent temporal bins. By default, a Student-t prior distribution (like a Gaussian but with heavier tails) is placed on the log of these ratios. This results in a prior SFH that tends toward constant SFR, and down-weights drmamtic changes in the SFR between adjacent bins. The overall normalization is provided by the `logmass` parameter.

In detail, the SFR in each timetime is computed as

$$\mathrm{SFR}_i = K \prod_{j=1}^{j<i} r_j$$

where K is a normalization constant. These are then converted to masses by multiplication with the bin widths and renormalization by the total mass.

To change the number of bins see `prospect.models.templates.adjust_continuity_agebins()`. This method produces 3 bins with defined edges at recent and very distant lookback times, and then divides the remaining time in to bins of equal intervals of $\log(t_{\mathrm{lookback}})$

5.3.2 Continuity Flex SFH

See leja19 for more details. A set of prospector parameters implementing this treatment is available as the `"continuity_flex_sfh"` entry of `prospect.models.templates.TemplateLibrary`

In this parameterization, the edges of the temporal bins are adjusted such that for a given set of SFRs an equal amount of mass forms in each bin. In other words, the bins all contain the same fraction of the total stellar mass, and the free parameters are related to the time it takes each succesive quantile of the mass to form. The widths are derived from the J sampled SFR ratios $r_j = \mathrm{SFR}_j/\mathrm{SFR}_{j+1}$ as

$$\Delta t_0 = t_{\mathrm{flex}}/(1 + \sum_{n=1}^{n=J} \prod_{j=1}^{j=n} r_j)$$

$$\Delta t_i = \Delta t_0 \prod_{j=1}^{j=i} r_j$$

where t is lookback time. Note that the width of the first and last bin are fixed to the values supplied in the initial `"agebins"` parameter, while $t_{\rm flex}$ is the remaining interval of lookback time.

5.3.3 PSB Hybrid SFH

See suess21 for details.

This parameterization provides a number of fixed width bins at both small and large lookback times, combined with a number of flexible width bins between these fixed bins. These are designed to efficiently produce the flexibility required to model post-starburst SFHs. A set of prospector parameters implementing this treatment is available as the `"continuity_psb_sfh"` entry of `prospect.models.templates.TemplateLibrary`

5.3.4 Dirichlet SFH

See leja17, leja19 for more details. A set of prospector parameters implementing this treatment is available as the `"dirichlet_sfh"` entry of `prospect.models.templates.TemplateLibrary`

In this parameterization the sampling variables are related to the fraction of the total stellar mass formed in each bin. Since these fractions must add up to 1, the parameter space corresponds to a Dirichlet distribution, and for numerical reasons this is best represented by sampling in a dimensionless vector variable `z_fraction` with a specific prior distribution. Transformations from these dimensionless variables to SFRs or masses in each bin are provided in *prospect.models.transforms*.

CHAPTER

SIX

NEBULAR EMISSION

Nebular emission in prospector is based on the implementation in FSPS, which uses cloudy photoionization predictions for emission lines and nebular continuum with FSPS stellar populations as the ionization sources, as described in byler17.

6.1 FSPS Nebular Emission Parameters

The fundamental parameters of the nebular emission model are the ionization parameter U (`"gas_logu"`) and the gas-phase metallicity (`"gas_logz"`). In FSPS is possible to turn on or off nebular emission (line and continuum) using the `"add_neb_emission"` switch. It is also possible to turn off only the nebular continuum with the `"add_neb_continuum"` switch. By default FSPS will add the emission lines to the model spectrum internally. In some situations (see below) this is not desired, and so the `"nebemlineinspec"` switch can be used to keep the lines from being added – though their luminosities will still be computed by FSPS – in which case prospector can be used to add the lines to the predicted spectra and photometry.

The nebular emission grids were computed for ionization sources (stellar populations) with the same metallicity as the gas-phase, so to keep perfect consistency it is necessary to tie these parameters together.

6.2 Fitting Emission Line Fluxes

The nebular emission line grids available in FSPS may not be flexible enough to fit all the emission line ratios observed in nature (e.g. due to AGN actvitiy, shocks, or radiative transfer, photoionization, and abundance effects not captured in the cloudy modeling). Nevertheless, it may be desirable to have accurate models for the emission lines, e.g. to account for their presence in the center of an absorption line of interest. For this reason it is possible with prospector, when fitting spectroscopic data, to *fit* for the true emission line fluxes and compute model likelihoods marginalized over the possible emission line fluxes. The methodology is described in johnson21, and is enabled via the `"marginalize_elines"` model parameter.

Briefly, explicit parameters are introduced to account for the emission line widths (`"eline_sigma"`) and redshift offset from the stellar model (`"eline_delta_zred"`). These default to 100 km/s and the systemic redshift respectively, but can be adjusted or sampled and can be the same for every line (scalar) or give the value separately for every line (vector). The maximum likelihood emission line fluxes and the uncertainties thereon can then be determined from linear least-squares. Priors based on the cloudy-FSPS predictions can also be incorporated.

Note that this yields emission line ratios that are no longer tied to a physical model. It is possible to use the cloudy-FSPS predictions of the line luminosities for *some* lines while fitting for others, and this is described below.

6.3 Choosing Lines to Fit, Fix, or Ignore

Several additional parameters can be used to decide which lines to fit and marginalize over and which lines to include with luminosities set to the cloudy-FSPS values. It is also possible to completely ignore the contribution of specific lines (i.e. to not include them in the modeled spectrum or photometry). These parameters are:

- **"elines_to_fit" A list of lines to fit via linear-least-squares and** marginalize over. If this parameter is not given but "marginalize_elines" is True, all lines within the observed spectral range will be fit.
- **"elines_to_fix" A list of lines to fix to their cloudy+FSPS predicted** luminosities.
- **"elines_to_ignore" A list of lines to ignore in the spectral and** photometric models. Thier luminosities are still computed, and can be accessed through the prospect.models.sedmodel.SpecModel object at each likelihood call. This parameter takes effect regardless of the presence of spectral data or the value of "marginalize_elines"

In all cases the line names to use in these lists are those given in the FSPS emission line line information table, $SPS_HOME/data/emlines_info.dat, e.g. "Ly alpha 1216" for the Lyman-alpha 1216 Angstrom line.

6.4 Nebular Parameter Templates

Several default model parameter templates are available in prospect.models.templates.TemplateLibrary to easily include different emission line treatments in prospector modeling including properly setting the values of the various switches described above.

- **"nebular" A basic parameter set in which the nebular emission is based** on the cloudy-FSPS grids, the lines are added to the model within FSPS, and the gas-phase metallicity is tied to the stellar metallicity. The only free parameter introduced by this template is "gas_logu".
- **"nebular_marginalization" A parameter set for fitting and** marginalizing over the emission line luminosities, with a prior based on the cloudy-fsps predictions. By default all lines will be fit as long as FSPS is installed, and the line widths are included as a free parameter with uniform prior. This parameter set adds one new free parameter, "eline_sigma".
- **"fit_eline_redshift" This template can be used with** "nebular_marginaliztion" above to also fit for the redshift offset of the emission lines from the stellar model. It adds one free parameter, "eline_delta_zred".

CHAPTER

SEVEN

OUTPUT FORMAT

By default the output of the code is an HDF5 file, with filename `<output>_<timestamp>_mcmc.h5`

Optionally several pickle files (pickle is Python's internal object serialization module), roughly equivalent to IDL SAVE files, can be output. These may be convenient, but are not very portable.

7.1 HDF5 output

The output HDF5 file contains `datasets` for the input observational data and the MCMC sampling chains. A significant amount of metadata is stored as JSON in dataset attributes. Anything that could not be JSON serialized during writing will have been pickled instead, with the pickle stored as string data in place of the JSON.

The HDF5 files can read back into python using

```
import prospect.io.read_results as reader
filename = "<outfilestring>_<timestamp>_mcmc.h5"
results, obs, model = reader.results_from(filename)
```

which gives a `results` dictionary, the `obs` dictionary containing the data to which the model was fit, and the `model` object used in the fitting. The `results` dictionary contains the production MCMC chains from *emcee* or the chains and weights from *dynesty*, basic descriptions of the model parameters, and the `run_params` dictionary. Some additional ancillary information is stored, such as code versions, runtimes, MCMC acceptance fractions, and model parameter positions at various phases of of the code. There is also a string version of the **parameter file** used. The results dictionary contains the information needed to regenerate the *sps* object used in generating SEDs.

```
sps = reader.get_sps(res)
```

It can sometimes be difficult to reconstitute the model object if it is complicated, for example if it was built by referencing files or data that are no longer available. For this reason it is suggested that references to filenames in parameter files be made through command-line arguments that can be altered easily when reconsitituting the model.

7.2 Basic diagnostic plots

For detailed plotting, see the `prospect.plotting` module. Several methods for basic visualization of the results are also included in the `prospect.io.read_results` module.

First, the results file can be read into useful dictionaries and objects using :py:meth:prospect.io.read_results.results_from``

```
import prospect.io.read_results as reader
filename = "<outfilestring>_<timestamp>_mcmc"
results, obs, model = reader.results_from(filename)
```

It is often desirable to plot the parameter traces for the MCMC chains. That is, one wants to see the evolution of the parameter values as a function of MCMC iteration. This can be useful to check for convergence. It can be done easily for both *emcee* and *dynesty* results by

```
tracefig = reader.traceplot(results)
```

Another useful thing is to look at the "corner plot" of the parmeters. If one has the corner.py package, then

```
cornerfig = reader.subcorner(results, showpars=model.theta_labels()[:5])
```

will return a corner plot of the first 5 free parameters of the model. If `showpars` is omitted then all free parameters will be plotted. There are numerous other options to the `prospect.io.read_results.subcorner()` method, which is a thin wrapper on *corner.py*.

Finally, one often wants to look at posterior samples in the space of the data, or perhaps the maximum a posteriori parameter values. Taking the MAP as an example, this would be accomplished by

```
import numpy as np
# Find the index of the maximum a posteriori sample
ind_max = results["lnprobability"].argmax()
if res["chain"].ndim > 2:
    # emcee
    walker, iteration = np.unravel_index(ind_max, results["lnprobability"].shape)
            theta_max = results["chain"][walker, iteration, :]
elif res["chain"].ndim == 2:
    # dynesty
    theta_max = results["chain"][indmax, :]

# We need the SPS object to generate a model
sps = reader.get_sps(results)
# now generate the SED for the max. a post. parameters
spec, phot, x = model.predict(theta_max, obs=obs, sps=sps)

# Plot the data and the MAP model on top of each other
import matplotlib.pyplot as pl
if obs['wave'] is None:
            wave = sps.wavelengths
else:
    wave = obs['wavelength']
pl.plot(wave, obs['spectrum'], label="Spec Data")
pl.plot(wave, spec, label="MAP model spectrum")
if obs['filters'] is not None:
    pwave = [f.wave_effective for f in obs["filters"]]
    pl.plot(pwave, obs['maggies'], label="Phot Data")
    pl.plot(pwave, phot, label="MAP model photometry")
```

However, if all you want is the MAP model this may be stored for you, without the need to regenerate the `sps` object

```
import matplotlib.pyplot as pl
        best = res["bestfit"]
```

(continues on next page)

(continued from previous page)

```
a = model.params["zred"] + 1
pl.plot(a * best["restframe_wavelengths"], best['spectrum'], label="MAP spectrum")
if obs['filters'] is not None:
    pwave = [f.wave_effective for f in obs["filters"]]
    pl.plot(pwave, best['photometry'], label="MAP photometry")
```

CHAPTER

EIGHT

QUICKSTART

Here's a quick intro that fits 5-band SDSS photometry with a simple delay-tau parametric SFH model. This assumes you've successully installed prospector and all the prerequisites. This is intended simply to introduce the key ingredients; for more realistic usage see *Demonstrations* or the *Tutorial*.

```
import fsps
import dynesty
import sedpy
import h5py, astropy
import numpy as np
import astroquery
```

8.1 Build an observation

First we'll get some data, using `astroquery` to get SDSS photometry of a galaxy. We'll also get spectral data so we know the redshift.

```
from astroquery.sdss import SDSS
from astropy.coordinates import SkyCoord
bands = "ugriz"
mcol = [f"cModelMag_{b}" for b in bands]
ecol = [f"cModelMagErr_{b}" for b in bands]
cat = SDSS.query_crossid(SkyCoord(ra=204.46376, dec=35.79883, unit="deg"),
                         data_release=16,
                         photoobj_fields=mcol + ecol + ["specObjID"])
shdus = SDSS.get_spectra(plate=2101, mjd=53858, fiberID=220)[0]
assert int(shdus[2].data["SpecObjID"][0]) == cat[0]["specObjID"]
```

Now we will put this data in a dictionary with format expected by prospector. We convert the magnitudes to maggies, convert the magnitude errors to flux uncertainties (including a noise floor), and load the filter transmission curves using *sedpy*. We'll store the redshift here as well for convenience. Note that for this example we do *not* attempt to fit the spectrum at the same time.

```
from sedpy.observate import load_filters
from prospect.utils.obsutils import fix_obs

filters = load_filters([f"sdss_{b}0" for b in bands])
maggies = np.array([10**(-0.4 * cat[0][f"cModelMag_{b}"]) for b in bands])
magerr = np.array([cat[0][f"cModelMagErr_{b}"] for b in bands])
magerr = np.clip(magerr, 0.05, np.inf)
```

(continues on next page)

(continued from previous page)

```
obs = dict(wavelength=None, spectrum=None, unc=None, redshift=shdus[2].data[0]["z"],
           maggies=maggies, maggies_unc=magerr * maggies / 1.086, filters=filters)
obs = fix_obs(obs)
```

8.2 Build a Model

Here we will get a default parameter set for a simple parametric SFH, and add a set of parameters describing nebular emission. We'll also fix the redshift to the value given by SDSS. This model has 5 free parameters, each of which has a an associated prior distribution. These default prior distributions can and should be replaced or adjusted depending on your particular science question.

```
from prospect.models.templates import TemplateLibrary
from prospect.models import SpecModel
model_params = TemplateLibrary["parametric_sfh"]
model_params.update(TemplateLibrary["nebular"])
model_params["zred"]["init"] = obs["redshift"]

model = SpecModel(model_params)
assert len(model.free_params) == 5
print(model)
```

In principle we could also add noise models for the spectral and photometric data, but we'll make the default assumption of independent Gaussian noise for the moment.

```
noise_model = (None, None)
```

8.3 Get a 'Source'

Now we need an object that will actually generate the galaxy spectrum using stellar population synthesis. For this we will use an object that wraps FSPS allowing access to all the parameterized SFHs. We will also just check which spectral and isochrone librariews are being used.

```
from prospect.sources import CSPSpecBasis
sps = CSPSpecBasis(zcontinuous=1)
print(sps.ssp.libraries)
```

8.4 Make a prediction

We can now predict our data for any set of parameters. This will take a little time because fsps is building and caching the SSPs. Subsequent calls to predict will be faster. Here we'll just make the predicition for the current value of the free parameters.

```
current_parameters = ",".join([f"{p}={v}" for p, v in zip(model.free_params, model.
↪theta)])
print(current_parameters)
```

(continues on next page)

(continued from previous page)

```
spec, phot, mfrac = model.predict(model.theta, obs=obs, sps=sps)
print(phot / obs["maggies"])
```

8.5 Run a fit

Since we can make predictions and we have data and uncertainties, we should be able to construct a likelihood function. Here we'll use the pre-defined default posterior probability function. We also set some sampling related keywords to make the fit go a little faster, though it should still take of order tens of minutes.

```
from prospect.fitting import lnprobfn, fit_model
fitting_kwargs = dict(nlive_init=400, nested_method="rwalk", nested_target_n_
↪effective=1000, nested_dlogz_init=0.05)
output = fit_model(obs, model, sps, optimize=False, dynesty=True, lnprobfn=lnprobfn,␣
↪noise=noise_model, **fitting_kwargs)
result, duration = output["sampling"]
```

The `result` is a dictionary with keys giving the Monte Carlo samples of parameter values and the corresponding posterior probabilities. Because we are using dynesty, we also get weights associated with each parameter sample in the chain.

Typically we'll want to save the fit information. We can save the output of the sampling along with other information about the model and the data that was fit as follows:

```
from prospect.io import write_results as writer
hfile = "./quickstart_dynesty_mcmc.h5"
writer.write_hdf5(hfile, {}, model, obs,
                  output["sampling"][0], None,
                  sps=sps,
                  tsample=output["sampling"][1],
                  toptimize=0.0)
```

8.6 Make plots

Now we'll want to read the saved fit information and make plots. To read the information we can use the built-in reader.

```
from prospect.io import read_results as reader
hfile = "./quickstart_dynesty_mcmc.h5"
out, out_obs, out_model = reader.results_from(hfile)
```

This gives a dictionary of useful information (`out`), as well as the obs dictionary that we were using and, in some cases, a reconsitituted model object. However, that is only possible if the model generation code is saved to the results file, in the form of the text for a *build_model()* function. Here we will use just use the model object that we've already generated.

Now we will do some plotting. First, lets make a corner plot of the posterior. We'll mark the highest probablity posterior sample as well.

```
import matplotlib.pyplot as pl
from prospect.plotting import corner
```

(continues on next page)

(continued from previous page)

```
nsamples, ndim = out["chain"].shape
cfig, axes = pl.subplots(ndim, ndim, figsize=(10,9))
axes = corner.allcorner(out["chain"].T, out["theta_labels"], axes, weights=out["weights
↪"], color="royalblue", show_titles=True)

from prospect.plotting.utils import best_sample
pbest = best_sample(out)
corner.scatter(pbest[:, None], axes, color="firebrick", marker="o")
```

Note that the highest probability sample may well be different than the peak of the marginalized posterior distribution.

Now let's plot the observed SED and the spectrum and SED of the highest probability posterior sample.

```
import matplotlib.pyplot as pl
sfig, saxes = pl.subplots(2, 1, gridspec_kw=dict(height_ratios=[1, 4]), sharex=True)
ax = saxes[1]
pwave = np.array([f.wave_effective for f in out_obs["filters"]])
# plot the data
ax.plot(pwave, out_obs["maggies"], linestyle="", marker="o", color="k")
ax.errorbar(pwave,  out_obs["maggies"], out_obs["maggies_unc"], linestyle="", color="k",␣
↪zorder=10)
ax.set_ylabel(r"$f_\nu$ (maggies)")
ax.set_xlabel(r"$\lambda$ (AA)")
ax.set_xlim(3e3, 1e4)
ax.set_ylim(out_obs["maggies"].min() * 0.1, out_obs["maggies"].max() * 5)
ax.set_yscale("log")

# get the best-fit SED
bsed = out["bestfit"]
ax.plot(bsed["restframe_wavelengths"] * (1+out_obs["redshift"]), bsed["spectrum"], color=
↪"firebrick", label="MAP sample")
ax.plot(pwave, bsed["photometry"], linestyle="", marker="s", markersize=10, mec="orange",
↪ mew=3, mfc="none")

ax = saxes[0]
chi = (out_obs["maggies"] - bsed["photometry"]) / out_obs["maggies_unc"]
ax.plot(pwave, chi, linestyle="", marker="o", color="k")
ax.axhline(0, color="k", linestyle=":")
ax.set_ylim(-2, 2)
ax.set_ylabel(r"$\chi_{\rm best}$")
```

Sometimes it is desirable to reconstitute the SED from a particular posterior sample or set of samples, or even the spectrum of the highest probability sample if it was not saved. This requires both the model and the sps object generated previously.

```
from prospect.plotting.utils import sample_posterior
# Here we fairly and randomly choose a posterior sample
p = sample_posterior(out["chain"], weights=out["weights"], nsample=1)
# show this sample in the corner plot
corner.scatter(p.T, axes, color="darkslateblue", marker="o")
# regenerate the spectrum and plot it
spec, phot, mfrac = model.predict(p[0], obs=out_obs, sps=sps)
ax = saxes[1]
```

(continues on next page)

(continued from previous page)

```
ax.plot(sps.wavelengths * (1+out_obs["redshift"]), spec, color="darkslateblue", label=
↪"posterior sample")
```

CHAPTER

NINE

DEMONSTRATIONS

You can check out the Jupyter notebook demo at the InteractiveDemo

Code used to make the fits and figures in the prospector paper is available in this github repository.

9.1 Interactive Figure

Also, below is an example of inference from an increasing number of photometric bands. Model parameters and SEDs are inferred (in blue) from a changing number of mock photometric bands (grey points). The mock is generated at the parameters and with the SED marked in black. This shows how with a small amount of data most posteriors are determined by the prior (dotted green) but that as the number of bands increases, the data are more infomative and the posterior distributions are narrower than the prior.

Click the buttons show the inference for different filter sets:

CHAPTER

TEN

TUTORIAL

Here is a guide to running Prospector fits from the command line using parameter files, and working with the output. This is a generalization of the techniques demonstrated in the quickstart, with more detailed descriptions of how each of the ingredients works.

We assume you have installed Prospector and all its dependencies as laid out in the docs. The next thing you need to do is make a temporary work directory, `<workdir>`

```
cd <workdir>
cp <codedir>/demo/demo_* .
```

We now have a *parameter file* or two, and some data. Take a look at the `demo_photometry.dat` file in an editor, you'll see it is a simple ascii file, with a few rows and several columns. Each row is a different galaxy, each column is a different piece of information about that galaxy.

This is just an example. In practice Prospector can work with a wide variety of data types.

10.1 The parameter file

Open up `demo_params.py` in an editor, preferably one with syntax highlighting. You'll see that it's a python file. It includes some imports, a number of methods that build the ingredients for the fitting, and then an executable portion.

Executable Script

The executable portion of the parameter file that comes after the `if __name__ == "__main__"` line is run when the parameter file is called. Here the possible command line arguments and their default values are defined, including any custom arguments that you might add. In this example we have added several command line arguments that control how the data is read and how the The supplied command line arguments are then parsed and placed in a dictionary. This dictionary is passed to all the ingredient building methods (described below), which return the data dictionary and necessary model objects. The data dictionary and model objects are passed to a function that runs the prospector fit (`prospect.fitting.fit_model()`). Finally, the fit results are written to an output file.

Building the fit ingredients: build_model

Several methods must be defined in the parameter file to build the ingredients for the fit. The purpose of these functions and their required output are described here. You will want to modify some of these for your specific model and data. Note that each of these functions will be passed a dictionary of command line arguments. These command line arguments, including any you add to the command line parser in the executable portion of the script, can therefore be used to control the behaviour of the ingredient building functions. For example, a custom command line argument can be used to control the type of model that is fit, or how or from where the data is loaded.

First, the `build_model()` function is where the model that we will fit will be constructed. The specific model that you choose to construct depends on your data and your scientific question.

We have to specify a dictionary or list of model parameter specifications (see *Models*). Each specification is a dictionary that describes a single parameter. We can build the model by adjusting predefined sets of model parameter specifications, stored in the `models.templates.TemplateLibrary` dictionary-like object. In this example we choose the `"parametric_sfh"` set, which has the parameters necessary for a vasic delay-tau SFH fit with simple attenuation by a dust screen. This parameter set can be inspected in any of the following ways

```
from prospect.models.templates import TemplateLibrary, describe
# Show basic description of all pre-defined parameter sets
TemplateLibrary.show_contents()
# method 1: print the whole dictionary of dictionaries
model_params = TemplateLibrary["parametric_sfh"]
print(model_params)
# Method 2: show a prettier summary of the free and fixed parameters
print(describe(model_params))
```

You'll see that this model has 5 free parameters. Any parameter with `"isfree": True` in its specification will be varied during the fit. We have set priors on these parameters, visible as e.g. `model_params["mass"]["prior"]`. You may wish to change the default priors for your particular science case, using the prior objects in the `models.priors` module. An example of adjusting the priors for several parameters is given in the `build_model()` method in `demo_params.py`. Any free parameter *must* have an associated prior. Other parameters have their value set to the value of the `"init"` key, but do not vary during the fit. They can be made to vary by setting `"isfree": True` and specifying a prior. Parameters not listed here will be set to their default values. Typically this means default values in the `fsps.StellarPopulation` object; see python-fsps for details. Once you get a set of parameters from the `TemplateLibrary` you can modify or add parameter specifications. Since `model_params` is a dictionary (of dictionaries), you can update it with other parameter set dictionaries from the `TemplateLibrary`.

Finally, the `build_model()` function takes the `model_params` dictionary or list that you build and uses it to instantiate a `SedModel` object.

```
from prospect.models import SedModel
model_params = TemplateLibrary["parametric_sfh"]
# Turn on nebular emission and add associated parameters
model_params.update(TemplateLibrary["nebular"])
model_params["gas_logu"]["isfree"] = True
model = SedModel(model_params)
print(model)
```

If you wanted to change the specification of the model using custom command line arguments, you could do it in `build_model()` by allowing this function to take keyword arguments with the same name as the custom command line argument. This can be useful for example to set the initial value of the redshift `"zred"` on an object-by-object basis. Such an example is shown in `demo_params.py`, which also uses command line arguments to control whether nebular and/or dust emission parameters are added to the model.

Building the fit ingredients: build_obs

The next thing to look at is the `build_obs()` function. This is where you take the data from whatever format you have and put it into the dictionary format required by Prospector for a single object. This means you will have to modify this function heavily for your own use. But it also means you can use your existing data formats.

Right now, the `build_obs()` function just reads ascii data from a file, picks out a row (corresponding to the photometry of a single galaxy), and then makes a dictionary using data in that row. You'll note that both the datafile name and the object number are keyword arguments to this function. That means they can be set at execution time on the command line, by also including those variables in the `run_params` dictionary. We'll see an example later.

When you write your own `build_obs()` function, you can add all sorts of keyword arguments that control its output (for example, an object name or ID number that can be used to choose or find a single object in your data file). You can also import helper functions and modules. These can be either things like astropy, h5py, and sqlite or your own project

specific modules and functions. As long as the output dictionary is in the right format (see dataformat.rst), the body of this function can do anything.

Building the fit ingredients: the rest

Ok, now we go to the `build_sps()` function. This one is pretty straightforward, it simply instantiates our `sources.CSPSpecBasis` object. For nonparameteric fits one would use the `sources.FastStepBasis` object. These objects hold all the spectral libraries and produce an SED given a set of parameters. After that is `build_noise()`, which is for complexifying the noise model – ignore that for now.

10.2 Running a fit

There are two kinds of fitting packages that can be used with Prospector. The first is `emcee` which implements ensemble MCMC sampling, and the second is `dynesty`, which implements dynamic nested sampling. It is also possible to perform optimization. If `emcee` is used, the result of the optimization will be used to initalize the ensemble of walkers.

The choice of which fitting algorithms to use is based on command line flags (`--optimization`, `--emcee`, and `--dynesty`.) If no flags are set the model and data objects will be generated and stored in the output file, but no fitting will take place. To run the fit on object number 0 using `emcee` after an initial optimization, we would do the following at the command line

```
python demo_params.py --objid=0 --emcee --optimize \
--outfile=demo_obj0_emcee
```

If we wanted to change something about the MCMC parameters, or fit a different object, we could also do that at the command line

```
python demo_params.py --objid=1 --emcee --optimize \
--outfile=demo_obj1_emcee --nwalkers=32 --niter=1024
```

And if we want to use nested sampling with `dynesty` we would do the following

```
python demo_params.py --objid=0  --dynesty \
--outfile=demo_obj0_dynesty
```

Finally, it is sometimes useful to run the script from the interpreter to do some checks. This is best done with the IPython enhanced interactive python.

```
ipython
In [1]: %run demo_params.py --objid=0 --debug=True
```

You can then inspect the `obsdat` dictionary, the `model` object, and the `run_params` dictionary to make sure everything is working fine.

To see the full list of available command-line options, you can run the following

```
python demo_params.py --help
```

10.3 Working with the output

After the fit is completed we should have a file with a name like `demo_obj0_<fitter>_<timestamp>_mcmc.h5`. This is an HDF5 file containing sampling results and various configuration data, as well as the observational data that was fit. By setting `run_params["output_pickles"]=True` you can also output versions of this information in the less portable pickle format. We will read the HDF5 with python and make some plots using utilities in Prospector

To read the data back in from the output files that we've generated, use methods in `prospect.io.read_results`.

```
import prospect.io.read_results as reader
res, obs, model = reader.results_from("demo_obj_<fitter>_<timestamp>_mcmc.h5")
```

The `res` object is a dictionary containing various useful results. You can look at `res.keys()` to see a list of what it contains. The `obs` object is just the `obs` dictionary that was used in the fitting. The `model` object is the model object that was used in the fitting.

Diagnostic plots

There are also some methods in this module for basic diagnostic plots. The `subcorner` method requires that you have the corner package installed. It's possible now to examine the traces (i.e. the evolution of parameter value with MC iteration) and the posterior PDFs for the parameters.

```
# Trace plots
tfig = reader.traceplot(res)
# Corner figure of posterior PDFs
cfig = reader.subcorner(res)
```

Working with samples

If you want to get the *maximum a posteriori* sample, or percentiles of the posterior pdf, that can be done as follows (note that for `dynesty` the weights of each posterior sample must be taken into account when calculating quantiles) :

```
# Maximum posterior probability sample
imax = np.argmax(res['lnprobability'])
csz = res["chain"].shape
if res["chain"].ndim > 2:
    # emcee
    i, j = np.unravel_index(imax, res['lnprobability'].shape)
    theta_max = res['chain'][i, j, :].copy()
    flatchain = res["chain"].reshape(csz[0] * csz[1], csz[2])
        else:
    # dynesty
    theta_max = res['chain'][imax, :].copy()
    flatchain = res["chain"]

# 16th, 50th, and 84th percentiles of the posterior
from prospect.plotting.corner import quantile
weights = res.get("weights", None)
post_pcts = quantile(flatchain.T, q=[0.16, 0.50, 0.84], weights=weights)
```

Stored "best-fit" model

Further, the prediction of the data for the MAP posterior sample may be stored for you.

```
# Plot the stored maximum ln-probability sample
import matplotlib.pyplot as pl
```

(continues on next page)

(continued from previous page)

```
best = res["bestfit"]
a = model.params["zred"] + 1
pl.plot(a * best["restframe_wavelengths"], best['spectrum'], label="MAP spectrum")
if obs['filters'] is not None:
    pwave = [f.wave_effective for f in obs["filters"]]
    pl.plot(pwave, best['photometry'], label="MAP photometry")
    pl.set_title(best["parameter"])
```

This stored best-fit information is only available if the *sps* object was passed to the `write_hdf5()` after the fit is run. If it isn't available, you can regnerate the model predictions for the highest probability sample using the approach below.

Regenerating Model predictions

If necessary, one can regenerate models at any position in the posterior chain. This requires that we have the sps object used in the fitting to generate models, which we can regenerate using the `read_results.get_sps()` method.

```
# We need the correct sps object to generate models
sps = reader.get_sps(res)
```

Now we will choose a specific parameter value from the chain and plot what the observations and the model look like, as well as the uncertainty normalized residual. For `emcee` results we will use the last iteration of the first walker, while for `dynesty` results we will just use the last sample in the chain.

```
# Choose the walker and iteration number by hand.
walker, iteration = 0, -1
if res["chain"].ndim > 2:
    # if you used emcee for the inference
    theta = res['chain'][walker, iteration, :]
else:
    # if you used dynesty
    theta = res['chain'][iteration, :]

# Or get a fair sample from the posterior
from prospect.plotting.utils import sample_posterior
theta = sample_posterior(res["chain"], weights=res.get("weights", None), nsample=1)[0,:]

# Get the modeled spectra and photometry.
# These have the same shape as the obs['spectrum'] and obs['maggies'] arrays.
spec, phot, mfrac = model.predict(theta, obs=res['obs'], sps=sps)
# mfrac is the ratio of the surviving stellar mass to the formed mass (the ``"mass"``␣
↪parameter).

# Plot the model SED
import matplotlib.pyplot as pl
wave = [f.wave_effective for f in res['obs']['filters']]
sedfig, sedax = pl.subplots()
sedax.plot(wave, res['obs']['maggies'], '-o', label='Observations')
sedax.plot(wave, phot, '-o', label='Model at {},{}'.format(walker, iteration))
sedax.set_ylabel("Maggies")
sedax.set_xlabel("wavelength")
sedax.set_xscale('log')
```

(continues on next page)

(continued from previous page)

```
# Plot residuals for this walker and iteration
chifig, chiax = pl.subplots()
chi = (res['obs']['maggies'] - phot) / res['obs']['maggies_unc']
chiax.plot(wave, chi, 'o')
chiax.set_ylabel("Chi")
chiax.set_xlabel("wavelength")
chiax.set_xscale('log')
```

CHAPTER

ELEVEN

FREQUENTLY ASKED QUESTIONS

11.1 How do I add filter transmission curves?

Many projects use particular filter systems that are not general or widely used. It is easy to add a set of custom filter curves. Filter projections are handled by the sedpy code. See the FAQ there for detailed instructions on adding filter curves.

11.2 What units?

Prospector natively uses *maggies* for both spectra and photometry, and it is easiest if you supply data in these units. maggies are a flux density (f_ν) unit defined as Janskys/3631. Wavelengths are *vacuum* Angstroms by default. By default, masses are in solar masses of stars *formed*, i.e. the integral of the SFH. Note that this is different than surviving solar masses (due to stellar mass loss).

11.3 How long will it take to fit my data?

That depends. Here are some typical timings for each likelihood call in various situations (macbook pro)

- 10 band photometry, no nebular emission, no dust emission: 0.004s
- photometry, including nebular emission: 0.04s
- spectroscopy including FFT smoothing: 0.05s

Note that the initial likelihood calls will be (much) longer than this. Under the hood, the first time it is called python-FSPS computes and caches many quantities that are reused in subsequent calls.

Many likelihood evaluations may be required for the sampling to converge. This depends on the kind of model you are fitting and its dimensionality, the sampling algorithm being used, and the kind of data that you have. Hours or even days per fit is not uncommon for more complex models.

11.4 Can I fit my spectrum too?

There are several extra considerations that come up when fitting spectroscopy

1) Wavelength range and resolution. Prospector is based on FSPS, which uses the MILES spectral library. These have a resolution of ~2.5A FWHM from 3750AA - 7200AA restframe, and much lower (R~200 or so, but not actually well defined) outside this range. Higher resolution data (after including both velocity dispersion and instrumental resolution) or spectral points outside this range cannot yet be fit.

2) Relatedly, line spread function. Prospector includes methods for FFT based smoothing of the spectra, assuming a gaussian LSF (in either wavelength or velocity space). There is also the possibility of FFT based smoothing for wavelength dependent gaussian dispersion (i.e. sigma_lambda = f(lambda) with f possibly a polynomial of lambda). In practice the smoothed spectra will be a combination of the library resolution plus whatever FFT smoothing is applied. Hopefully this can be made to match your actual data resolution, which is a combination of the physical velocity dispersion and the instrumental resolution. The smoothing is controlled by the parameters *sigma_smooth*, *smooth_type*, and *fftsmooth*

3) Nebular emission. While prospector/FSPS include self-consistent nebular emission, the treatment is probably not flexible enough at the moment to fit high S/N, high resolution data including nebular emission (e.g. due to deviations of line ratios from Cloudy predictions or to complicated gas kinematics that are different than stellar kinematics). Thus fitting nebular lines should take adavantage of the nebular line amplitude optimization/marginalization capabilities. For very low resolution data this is less of an issue.

4) Spectrophotometric calibration. There are various options for dealing with the spectral continuum shape depending on how well you think your spectra are calibrated and if you also fit photometry to tie down the continuum shape. You can optimize out a polynomial "calibration" vector, or simultaneously fit for a polynomial and marginalize over the polynomial coefficients (this allows you to place priors on the accuracy of the spectrophotometric calibration). Or you can just take the spectrum as perfectly calibrated.

11.5 How do I fit for redshift as well as other parameters?

The simplest way is to just let the parameter specification for `"zred"` indicate that it is free and specify a prior for it (see *Models*). If you don't include the `"lumdist"` parameter at all in the `model_params` dictionary, then the luminosity distance will be computed directly from the redshift assuming a WMAP9 cosmology.

The other thing to keep in mind when the redshift is free is that the SFH might be constrained by the age of the universe at that redshift (e.g. `"tage"` should always be less than `t_univ(zred)`). If this is a concern, you can use parameter transformations (again, see *Models*). Specifically for parametric SFHs one would make the `"tage"` parameter fixed but *depend on* `prospect.models.transforms.tage_from_tuniv()` and add a new `"tage_tuniv"` parameter that corresponds to `"tage"` as a fraction of the age of the universe (with values from 0-1).

11.6 What SFH parameters should I use?

That depends on the scientific question you are trying to answer, and to some extent on the data that you have.

11.7 How do I use the non-parametric SFHs?

Prospector is packaged with four families of non-parametric star formation histories. The simplest model fits for the (log or linear) mass formed in fixed time bins. This is the most flexible model but typically results in unphysical priors on age and specific star formation rates. Another model is the Dirichlet parameterization introduced in leja17, in which the fractional specific star formation rate for fixed each time bin follows a Dirichlet distribution. This model is moderately flexible and produces reasonable age and sSFR priors, but it still allows unphysical SFHs with sharp quenching and rejuvenation events. A third model, the continuity prior, trades some flexibility for accuracy by explicitly weighting against these (typically considered unphysical) sharply quenching and rejuvenating SFHs. This is done by placing a prior on the ratio of SFRs in adjacent time bins to ensure a smooth evolution of SFR(t). Finally, the flexible continuity prior retains this smoothness weighting but instead of fitting for the mass forms in fixed bins, fits for the size of time bins in which a fixed amount of mass is formed. This prior removes the discretization effect of the time bins in exchange for imposing a minimum mass resolution in the recovered SFH parameters. The performance of these different nonparametric models is compared and contrasted in detail in leja19.

In order to use these models, select the appropriate *parameter set template* to use in the `model_params` dictionary. You will also need to make sure to use the appropriate *source* object, `prospect.sources.FastStepBasis`.

The parameter templates are set up to transform from the sampling parameters (e.g. `logsfr_ratios`) to the fundamental parameters of the non-parametric SFH, the temporal bins and vector of masses formed in each bin. To change the bin widths or number of bins, several related aspects of the parameter set including the length of several parameters and the priors must be changed simultaneously. See `prospect.models.templates.adjust_continuity_agebins()` for an example.

11.8 What bins should I use for the non-parametric SFH?

Deciding on the "optimal" number of bins to use in such non-parametric SFHs is a difficult question. The pioneering work of ocvirk06 suggests approximately 10 independent components can be recovered from extremely high S/N R=10000 spectra (and perfect models). The fundamental problem is that the spectra of single age populations change slowly with age (or metallicity), so the contributions of each SSP to a composite spectrum are very highly degenerate and some degree of regularization or prior information is required. However, the ideal regularization depends on the (*a priori* unknown) SFH of the galaxy. For example, for a narrow burst one would want many narrow bins near the burst and wide bins away from it. Reducing the number of bins effectively amounts to collapsing the prior for the ratio of the SFR in two sub-bins to a delta-function at 1. Using too few bins can result in biases in the same way as the strong priors imposed by parametric models. Tests in leja19 suggest that ~5 bins are adequate to model covariances in basic parameters from photometry, but more bins are better to explore detailed constraints on SFHs.

11.9 So should I use *emcee*, *nestle*, or *dynesty* for posterior sampling?

We recommend using the *dynesty* nested sampling package.

In addition to the standard sampling phase which terminates based on the quality of the estimation of the Bayesian evidence, *dynesty* includes a subsequent dynamic sampling phase which, as implemented in Prospector, instead terminates based the quality of the posterior estimation. This permits the user to specify stopping criteria based directly on the density of the posterior sampling with the `nested_target_n_effective` keyword, providing direct control over the trade-off between posterior quality and computational time. A value of 10,000 for this keyword specifies high-quality posteriors, whereas a value of 3,000 will produce reasonable but approximate posteriors. Additionally, *dynesty* sampling can be parallelized in Prospector: this produces faster convergence time at the cost of lower computational efficiency (i.e., fewer model evaluations per unit computational time). It is best suited for fast evaluation of small samples of objects, whereas single-core fits produce more computationally efficient fits to large samples of objects.

11.10 What settings should I use for *dynesty*?

The default *dynesty* settings in Prospector are optimized for a low-dimensional (N=4-7) model. Higher-dimensional models with more complex likelihood spaces will likely require more advanced *dynesty* settings to ensure efficient and timely convergence. This often entails increasing the number of live points, changing to more robust sampling methodology (e.g., from uniform to a random walk), setting a maximum number of function calls, or altering the target evidence and posterior thresholds. More details can be found in speagle20 and the dynesty online documentation. The list of options and their default values can be seen with

```
from prospect.utils import prospect_args
prospect_args.show_default_args()
```

11.11 The chains did not converge when using *dynesty*, why?

It is likely that they did converge; note that the convergence for MC sampling of a posterior PDF is not defined by the samples all tending toward the a single value, but as the *distribution* of samples remaining stable. The samples for a poorly constrained parameter will remain widely dispersed, even if the MC sampling has converged to the correct *distribution*

11.12 How do I use *emcee* in Prospector?

For each parameter, an initial value (`"init"` in the parameter specification) must be given. The ensemble of walkers is initialized around this value, with a Gaussian spread that can be specified separately for each parameter. Each walker position is evolved at each iteration using parameter proposals derived from an ensemble of the other walkers. In order to speed up initial movement of the cloud of walkers to the region of parameter space containing most of the probability mass, multiple user defined rounds of burn-in may be performed. After each round the walker distribution in parameter space is re-initialized to a multivariate Gaussian derived from the best 50% of the walkers (where best is defined in terms of posterior probability at the last iteration). The iterations in these burn-in rounds are discarded before a final production run. It is important to ensure that the chain of walkers has converged to a stable *distribution* of parameter values. Diagnosing convergence is fraught; a number of indicators have been proposed sharma17 including the auto-correlation time of the chain goodman10. Comparing the results of separate chains can also provide a sanity check.

11.13 When should I use optimization?

Optimization can be performed before ensemble MCMC sampling, to decrease the burn-in time of the MCMC algorithm. Prospector currently supports Levenburg-Marquardt least-squares optimization and Powell's method, as implemented in SciPy. It is possible to start optimizations from a number of different parameter values, drawn from the prior parameter distribution, in order to mitigate the problems posed by local maxima.

Note that this optimization method requires that the number of data points (photometry or spectroscpy) be larger than the number of free model parameters.

11.14 How do I plot the best fit SED? How do I plot uncertainties on that?

Prospector can compute and store the SED prediction for the highest probability sample, in the `"bestfit"` group of the output HDF5 file.

Note that the highest probability sample is *not* the same as the maximum a posteriori (MAP) solution. The MAP solution inhabits a vanishingly small region of the prior parameter space; it is exceedingly unlikely that the MCMC sampler would visit exactly that location. Furthermore, when large degeneracies are present, the maximum a posteriori parameters may be only very slightly more likely than many solutions with very different parameters.

To plot uncertainties we recommend regenerating SED predictions for a fair sample from the posterior PDF and estimating quantiles of the flux at each wavelength.

11.15 How do I get the wavelength array for plotting spectra and/or photometry when fitting only photometry?

When fitting only photometry, the *restframe* wavelength array for the predicted spectrum can be found in the `wavelengths` attribute of *`prospect.sources.SSPBasis`*. The wavelengths of the filters can be obtained from the `wave_effective` attribute of each filter in the `obs["filters"]` list.

11.16 Should I fit spectra in the restframe or the observed frame?

You can do either if you are fitting only spectra. If fitting in the restframe then the distance has to be specified explicitly via a `lumdist` model parameter because otherwise it is inferred from the redshift which for restframe spectra is 0.

If you are fitting photometry and spectroscopy simultaneously then you should be fitting the observed frame spectra.

11.17 How do I obtain posteriors for the surviving stellar mass instead of the formed stellar mass

By default the units of stellar mass used in prospector models are the *formed* stellar mass. This is different than the 'current' or surviving stellar mass due to stellar mass loss during evolution (e.g. AGB winds, supernovae) in a way that depends on metallicity, SFH, and IMF. The ratio between the surviving stellar stellar mass and the formed stellar mass (often referred to in the code as `mfrac` is returned by by the `prospect.models.SpecModel.predict()` method, and the surviving stellar mass can be obtained for any given parameter set as:

```
spec, phot, mfrac = model.predict(parameter_vector, obs=obs, sps=sps)
surviving_mass = np.sum(model.params["mass"]) * mfrac
```

When fitting parametric SFH models using *`prospect.sources.CSPSpecBasis`* it may be possible to fit directly in surviving stellar mass by adding the following fixed parameter to your model specification:

```
model_params["mass_units"]=dict(init="mstar", isfree=False, N=1)
```

Note that the surviving stellar mass will include stellar remnants (black holes, neutron stars, and white dwarfs) by default. This can be controlled via the (FSPS) parameter `"add_stellar_remnants"`

11.18 What priors should I use?

That depends on the scientific question and the objects under consideration. In general we recommend using informative priors (e.g. narrow `Normal` distributions) for parameters that you think might matter at all.

11.19 What happens if a parameter is not well constrained? When should I fix parameters?

If some parameter is completely unconstrained you will get back the prior. There are also (often) cases where you are "prior-dominated", i.e. the posterior is mostly set by the prior but with a small perturbation due to small amounts of information supplied by the data. You can compare the posterior to the prior, e.g. using the Kullback-Liebler divergence between the two distributions, to see if you have learned anything about that parameter. Or just overplot the prior on the marginalized pPDFs

To be fully righteous you should only fix parameters if

- you are very sure of their values;
- or if you don't think changing the parameter will have a noticeable effect on the model;
- or if a parameter is perfectly degenerate (in the space of the data) with another parameter.

In practice parameters that have only a small effect but take a great deal of time to vary are often fixed.

11.20 What do I do about upper limits?

Ideally you will have flux measurements and associated Gaussian uncertainty for every filter or wavelength, even if the measurement is of a negative value. Properly accounting for upper limits involves a somewhat complicated adjustement to the likelihood function (see Appendix A here), but a reasonable approximation can be made by setting the flux to zero and the uncertainty to the 1-sigma upper limit.

11.21 What do I do with the chain? What values should I report?

This is a general question for MC sampling techniques. See sharma17 or speagle19 for advice.

11.22 Why isn't the posterior PDF centered on the highest posterior probability sample?

11.23 How do I interpret the *lnprobability* or *lnp* values? Why do I get *lnp* > *0*?

11.24 How do I know if Prospector is "working"?

CHAPTER

TWELVE

PROSPECT.MODELS

12.1 prospect.models

12.2 prospect.models.priors

priors.py – This module contains various objects to be used as priors. When called these return the ln-prior-probability, and they can also be used to construct prior transforms (for nested sampling) and can be sampled from.

class prospect.models.priors.**Prior**(*parnames=[]*, *name=''*, ***kwargs*)

Encapsulate the priors in an object. Each prior should have a distribution name and optional parameters specifying scale and location (e.g. min/max or mean/sigma). These can be aliased at instantiation using the `parnames` keyword. When called, the argument should be a variable and the object should return the ln-prior-probability of that value.

```
ln_prior_prob = Prior(param=par)(value)
```

Should be able to sample from the prior, and to get the gradient of the prior at any variable value. Methods should also be avilable to give a useful plotting range and, if there are bounds, to return them.

Parameters **parnames** (*sequence of strings*) – A list of names of the parameters, used to alias the intrinsic parameter names. This way different instances of the same Prior can have different parameter names, in case they are being fit for....

params

The values of the parameters describing the prior distribution.

Type dictionary

update(***kwargs*)

Update `self.params` values using alias.

sample(*nsample=None*, ***kwargs*)

Draw a sample from the prior distribution.

Parameters **nsample** – (optional) Unused

unit_transform(*x*, ***kwargs*)

Go from a value of the CDF (between 0 and 1) to the corresponding parameter value.

Parameters **x** – A scalar or vector of same length as the Prior with values between zero and one corresponding to the value of the CDF.

Returns theta The parameter value corresponding to the value of the CDF given by *x*.

inverse_unit_transform(*x*, ***kwargs*)

Go from the parameter value to the unit coordinate using the cdf.

property loc

This should be overridden.

property scale

This should be overridden.

12.3 prospect.models.transforms

transforms.py – This module contains parameter transformations that may be useful to transform from parameters that are easier to _sample_ in to the parameters required for building SED models.

They can be used as "depends_on" entries in parameter specifications.

prospect.models.transforms.**stellar_logzsol**(*logzsol=0.0*, ***extras*)

Simple function that takes an argument list and returns the value of the *logzsol* argument (i.e. the stellar metallicity)

Parameters **logzsol** (*float*) – FSPS stellar metaliicity parameter.

Returns **logzsol** – The same.

Return type float

prospect.models.transforms.**delogify_mass**(*logmass=0.0*, ***extras*)

Simple function that takes an argument list including a *logmass* parameter and returns the corresponding linear mass.

Parameters **logmass** (*float*) – The log10(mass)

Returns **mass** – The mass in linear units

Return type float

prospect.models.transforms.**tburst_from_fage**(*tage=0.0*, *fage_burst=0.0*, ***extras*)

This function transfroms from a fractional age of a burst to an absolute age. With this transformation one can sample in `fage_burst` without worry about the case `tburst > tage`.

Parameters

- **tage** (*float, Gyr*) – The age of the host galaxy.
- **fage_burst** (*float between 0 and 1*) – The fraction of the host age at which the burst occurred.

Returns **tburst** – The age of the host when the burst occurred (i.e. the FSPS `tburst` parameter)

Return type float, Gyr

prospect.models.transforms.**tage_from_tuniv**(*zred=0.0*, *tage_tuniv=1.0*, ***extras*)

This function calculates a galaxy age from the age of the universe at `zred` and the age given as a fraction of the age of the universe. This allows for both `zred` and `tage` parameters without `tage` exceeding the age of the universe.

Parameters

- **zred** (*float*) – Cosmological redshift.

- **tage_tuniv** (*float between 0 and 1*) – The ratio of `tage` to the age of the universe at `zred`.

Returns **tage** – The stellar population age, in Gyr

Return type float

`prospect.models.transforms.`**zred_to_agebins**(*zred=0.0*, *agebins=[]*, ***extras*)

Set the nonparameteric SFH age bins depending on the age of the universe at `zred`. The first bin is not altered and the last bin is always 15% of the upper edge of the oldest bin, but the intervening bins are evenly spaced in log(age).

Parameters

- **zred** (*float*) – Cosmological redshift. This sets the age of the universe.
- **agebins** (ndarray of shape `(nbin, 2)`) – The SFH bin edges in log10(years).

Returns **agebins** – The new SFH bin edges.

Return type ndarray of shape `(nbin, 2)`

`prospect.models.transforms.`**dustratio_to_dust1**(*dust2=0.0*, *dust_ratio=0.0*, ***extras*)

Set the value of dust1 from the value of dust2 and dust_ratio

Parameters

- **dust2** (*float*) – The diffuse dust V-band optical depth (the FSPS `dust2` parameter.)
- **dust_ratio** (*float*) – The ratio of the extra optical depth towards young stars to the diffuse optical depth affecting all stars.

Returns **dust1** – The extra optical depth towards young stars (the FSPS `dust1` parameter.)

Return type float

`prospect.models.transforms.`**logsfr_ratios_to_masses**(*logmass=None*, *logsfr_ratios=None*, *agebins=None*, ***extras*)

This converts from an array of log_10(SFR_j / SFR_{j+1}) and a value of log10(Sum_i M_i) to values of M_i. j=0 is the most recent bin in lookback time.

`prospect.models.transforms.`**logsfr_ratios_to_sfrs**(*logmass=None*, *logsfr_ratios=None*, *agebins=None*, ***extras*)

Convenience function

`prospect.models.transforms.`**logsfr_ratios_to_agebins**([*NBINS*, *2*])

use equation: delta(t1) = tuniv / (1 + SUM(n=1 to n=nbins-1) PROD(j=1 to j=n) Sn) where Sn = SFR(n) / SFR(n+1) and delta(t1) is width of youngest bin

`prospect.models.transforms.`**zfrac_to_masses**(*total_mass=None*, *z_fraction=None*, *agebins=None*, ***extras*)

This transforms from independent dimensionless z variables to sfr fractions and then to bin mass fractions. The transformation is such that sfr fractions are drawn from a Dirichlet prior. See Betancourt et al. 2010 and Leja et al. 2017

Parameters

- **total_mass** (*float*) – The total mass formed over all bins in the SFH.
- **z_fraction** (ndarray of shape `(Nbins-1,)`) – latent variables drawn from a specific set of Beta distributions. (see Betancourt 2010)

Returns **masses** – The stellar mass formed in each age bin.

Return type ndarray of shape `(Nbins,)`

`prospect.models.transforms.`**`zfrac_to_sfrac`**(*z_fraction=None*, ***extras*)

This transforms from independent dimensionless *z* variables to sfr fractions. The transformation is such that sfr fractions are drawn from a Dirichlet prior. See Betancourt et al. 2010 and Leja et al. 2017

Parameters **`z_fraction`** (ndarray of shape `(Nbins-1,)`) – latent variables drawn from a specific set of Beta distributions. (see Betancourt 2010)

Returns **sfrac** – The star formation fractions (See Leja et al. 2017 for definition).

Return type ndarray of shape `(Nbins,)`

`prospect.models.transforms.`**`zfrac_to_sfr`**(*total_mass=None*, *z_fraction=None*, *agebins=None*, ***extras*)

This transforms from independent dimensionless *z* variables to SFRs.

Returns **sfrs** The SFR in each age bin (msun/yr).

`prospect.models.transforms.`**`masses_to_zfrac`**(*mass=None*, *agebins=None*, ***extras*)

The inverse of *zfrac_to_masses()*, for setting mock parameters based on mock bin masses.

Returns

- **total_mass** (*float*) – The total mass
- **zfrac** (ndarray of shape `(Nbins-1,)`) – latent variables drawn from a specific set of Beta distributions. (see Betancourt 2010) related to the fraction of mass formed in each bin.

CHAPTER

THIRTEEN

PROSPECT.SOURCES

Classes in the *prospect.sources* module are used to instantiate **sps** objects. They are defined by the presence of a `get_spectrum()` method that takes a wavelength array, a list of filter objects, and a parameter dictionary and return a spectrum, a set of broadband fluxes, and a blob of ancillary information.

Most of these classes are a wrapper on `fsps.StellarPopulation` objects, and as such have a significant memory footprint. The parameter dictionary can include any `fsps` parameter, as well as parameters used by these classes to control redshifting, spectral smoothing, wavelength calibration, and other aspects of the model.

class prospect.sources.**CSPSpecBasis**(*zcontinuous=1*, *reserved_params=['sigma_smooth']*, *vactoair_flag=False*, *compute_vega_mags=False*, ***kwargs*)

A subclass of *SSPBasis* for combinations of N composite stellar populations (including single-age populations). The number of composite stellar populations is given by the length of the `"mass"` parameter. Other population properties can also be vectors of the same length as `"mass"` if they are independent for each component.

update(***params*)

Update the *params* attribute, making parameters scalar if possible.

update_component(*component_index*)

Pass params that correspond to a single component through to the fsps.StellarPopulation object.

Parameters **component_index** – The index of the component for which to pull out individual parameters that are passed to the fsps.StellarPopulation object.

get_galaxy_spectrum(***params*)

Update parameters, then loop over each component getting a spectrum for each and sum with appropriate weights.

Parameters **params** – A parameter dictionary that gets passed to the `self.update` method and will generally include physical parameters that control the stellar population and output spectrum or SED.

Returns wave Wavelength in angstroms.

Returns spectrum Spectrum in units of Lsun/Hz/solar masses formed.

Returns mass_fraction Fraction of the formed stellar mass that still exists.

class prospect.sources.**SSPBasis**(*zcontinuous=1*, *reserved_params=['tage', 'sigma_smooth']*, *interp_type='logarithmic'*, *flux_interp='linear'*, *mint_log=- 3*, *compute_vega_mags=False*, ***kwargs*)

This is a class that wraps the fsps.StellarPopulation object, which is used for producing SSPs. The `fsps.StellarPopulation` object is accessed as `SSPBasis().ssp`.

This class allows for the custom calculation of relative SSP weights (by overriding `all_ssp_weights`) to produce spectra from arbitrary composite SFHs. Alternatively, the entire `get_galaxy_spectrum` method can be

overridden to produce a galaxy spectrum in some other way, for example taking advantage of weight calculations within FSPS for tabular SFHs or for parameteric SFHs.

The base implementation here produces an SSP interpolated to the age given by `tage`, with initial mass given by `mass`. However, this is much slower than letting FSPS calculate the weights, as implemented in `FastSSPBasis`.

Furthermore, smoothing, redshifting, and filter projections are handled outside of FSPS, allowing for fast and more flexible algorithms.

Parameters **reserved_params** – These are parameters which have names like the FSPS parameters but will not be passed to the StellarPopulation object because we are overriding their functionality using (hopefully more efficient) custom algorithms.

update(***params*)

Update the parameters, passing the *unreserved* FSPS parameters through to the `fsps.StellarPopulation` object.

Parameters **params** – A parameter dictionary.

get_galaxy_spectrum(***params*)

Update parameters, then multiply SSP weights by SSP spectra and stellar masses, and sum.

Returns wave Wavelength in angstroms.

Returns spectrum Spectrum in units of Lsun/Hz/solar masses formed.

Returns mass_fraction Fraction of the formed stellar mass that still exists.

get_galaxy_elines()

Get the wavelengths and specific emission line luminosity of the nebular emission lines predicted by FSPS. These lines are in units of Lsun/solar mass formed. This assumes that *get_galaxy_spectrum* has already been called.

Returns ewave The *restframe* wavelengths of the emission lines, AA

Returns elum Specific luminosities of the nebular emission lines, Lsun/stellar mass formed

get_spectrum(*outwave=None*, *filters=None*, *peraa=False*, ***params*)

Get a spectrum and SED for the given params.

Parameters

- **outwave** – (default: None) Desired *vacuum* wavelengths. Defaults to the values in `sps.wavelength`.
- **peraa** – (default: False) If *True*, return the spectrum in erg/s/cm^2/AA instead of AB maggies.
- **filters** – (default: None) A list of filter objects for which you'd like photometry to be calculated.
- **params** – Optional keywords giving parameter values that will be used to generate the predicted spectrum.

Returns spec Observed frame spectrum in AB maggies, unless `peraa=True` in which case the units are erg/s/cm^2/AA.

Returns phot Observed frame photometry in AB maggies.

Returns mass_frac The ratio of the surviving stellar mass to the total mass formed.

property all_ssp_weights

Weights for a single age population. This is a slow way to do this!

class prospect.sources.**FastStepBasis**(*zcontinuous=1*, *reserved_params=['tage', 'sigma_smooth']*, *interp_type='logarithmic'*, *flux_interp='linear'*, *mint_log=- 3*, *compute_vega_mags=False*, ***kwargs*)

Subclass of *SSPBasis* that implements a "nonparameteric" (i.e. binned) SFH. This is accomplished by generating a tabular SFH with the proper form to be passed to FSPS. The key parameters for this SFH are:

- `agebins` - array of shape `(nbin, 2)` giving the younger and older (in lookback time) edges of each bin in log10(years)
- `mass` - array of shape `(nbin,)` giving the total stellar mass (in solar masses) **formed** in each bin.

get_galaxy_spectrum(***params*)

Construct the tabular SFH and feed it to the `ssp`.

convert_sfh(*agebins*, *mformed*, *epsilon=0.0001*, *maxage=None*)

Given arrays of agebins and formed masses with each bin, calculate a tabular SFH. The resulting time vector has time points either side of each bin edge with a "closeness" defined by a parameter epsilon.

Parameters

- **agebins** – An array of bin edges, log(yrs). This method assumes that the upper edge of one bin is the same as the lower edge of another bin. ndarray of shape `(nbin, 2)`
- **mformed** – The stellar mass formed in each bin. ndarray of shape `(nbin,)`
- **epsilon** – (optional, default 1e-4) A small number used to define the fraction time separation of adjacent points at the bin edges.
- **maxage** – (optional, default: `None`) A maximum age of stars in the population, in yrs. If `None` then the maximum value of `agebins` is used. Note that an error will occur if maxage < the maximum age in agebins.

Returns time The output time array for use with sfh=3, in Gyr. ndarray of shape (2*N)

Returns sfr The output sfr array for use with sfh=3, in M_sun/yr. ndarray of shape (2*N)

Returns maxage The maximum valid age in the returned isochrone.

class prospect.sources.**BlackBodyDustBasis**(***kwargs*)

get_spectrum(*outwave=None*, *filters=None*, ***params*)

Given a params dictionary, generate spectroscopy, photometry and any extras (e.g. stellar mass).

Parameters

- **outwave** – The output wavelength vector.
- **filters** – A list of sedpy filter objects.
- **params** – Keywords forming the parameter set.

Returns spec The restframe spectrum in units of erg/s/cm^2/AA

Returns phot The apparent (redshifted) maggies in each of the filters.

Returns extras A list of None type objects, only included for consistency with the SedModel class.

one_sed(*icomp=0*, *wave=None*, *filters=None*, ***extras*)

Pull out individual component parameters from the param dictionary and generate spectra for those components

`normalization()`

This method computes the normalization (due do distance dimming, unit conversions, etc.) based on the content of the params dictionary.

CHAPTER

FOURTEEN

PROSPECT.FITTING

CHAPTER

FIFTEEN

PROSPECT.IO

15.1 prospect.io.read_results

15.2 prospect.io.write_results

CHAPTER

SIXTEEN

PROSPECT.PLOTTING

16.1 prospect.plotting.utils

16.2 prospect.plotting.sfh

16.3 prospect.plotting.corner

CHAPTER

SEVENTEEN

PROSPECT.UTILS

17.1 prospect.utils.smoothing

prospect.utils.smoothing.**smoothspec**(*wave*, *spec*, *resolution=None*, *outwave=None*, *smoothtype='vel'*, *fftsmooth=True*, *min_wave_smooth=0*, *max_wave_smooth=inf*, ***kwargs*)

Parameters

- **wave** (ndarray of shape `(N_pix,)`) – The wavelength vector of the input spectrum. Assumed Angstroms.
- **spec** (ndarray of shape `(N_pix,)`) – The flux vector of the input spectrum.
- **resolution** (*float*) – The smoothing parameter. Units depend on `smoothtype`.
- **outwave** (`None` or ndarray of shape `(N_pix_out,)`) – The output wavelength vector. If `None` then the input wavelength vector will be assumed, though if `min_wave_smooth` or `max_wave_smooth` are also specified, then the output spectrum may have different length than `spec` or `wave`, or the convolution may be strange outside of `min_wave_smooth` and `max_wave_smooth`. Basically, always set `outwave` to be safe.
- **smoothtype** (*string, optional, default: "vel"*) – The type of smoothing to perform. One of:
  - `"vel"` - velocity smoothing, `resolution` units are in km/s (dispersion not FWHM)
  - `"R"` - resolution smoothing, `resolution` is in units of λ/σ_λ (where σ_λ is dispersion, not FWHM)
  - `"lambda"` - wavelength smoothing. `resolution` is in units of Angstroms
  - `"lsf"` - line-spread function. Use an aribitrary line spread function, which can be given as a vector the same length as `wave` that gives the dispersion (in AA) at each wavelength. Alternatively, if `resolution` is `None` then a line-spread function must be present as an additional `lsf` keyword. In this case all additional keywords as well as the `wave` vector will be passed to this `lsf` function.
- **fftsmooth** (*bool, optional, default: True*) – Switch to use FFTs to do the smoothing, usually resulting in massive speedups of all algorithms. However, edge effects may be present.
- **min_wave_smooth** (*float, optional default: 0*) – The minimum wavelength of the input vector to consider when smoothing the spectrum. If `None` then it is determined from the output wavelength vector and padded by some multiple of the desired resolution.

- **max_wave_smooth** (*float, optional, default: inf*) – The maximum wavelength of the input vector to consider when smoothing the spectrum. If None then it is determined from the output wavelength vector and padded by some multiple of the desired resolution.
- **inres** (*float, optional*) – If given, this parameter specifies the resolution of the input. This resolution is subtracted in quadrature from the target output resolution before the kernel is formed.

  In certain cases this can be used to properly switch from resolution that is constant in velocity to one that is constant in wavelength, taking into account the wavelength dependence of the input resolution when defined in terms of lambda. This is possible iff: * `fftsmooth` is False * `smoothtype` is `"lambda"` * The optional `in_vel` parameter is supplied and True.

  The units of `inres` should be the same as the units of `resolution`, except in the case of switching from velocity to wavelength resolution, in which case the units of `inres` should be in units of lambda/sigma_lambda.
- **in_vel** (*float (optional)*) – If supplied and True, the `inres` parameter is assumed to be in units of lambda/sigma_lambda. This parameter is ignored **unless** the `smoothtype` is `"lambda"` and `fftsmooth` is False.

Returns **flux** – The smoothed spectrum on the *outwave* grid, ndarray.

Return type ndarray of shape `(N_pix_out,)`

CHAPTER

EIGHTEEN

LICENSE AND ATTRIBUTION

If you use this code, please reference this paper:

```
@ARTICLE{2021ApJS..254...22J,
    author = {{Johnson}, Benjamin D. and {Leja}, Joel and {Conroy}, Charlie and {Speagle}
↪, Joshua S.},
        title = "{Stellar Population Inference with Prospector}",
    journal = {\apjs},
    keywords = {Galaxy evolution, Spectral energy distribution, Astronomy data modeling,␣
↪594, 2129, 1859, Astrophysics - Astrophysics of Galaxies, Astrophysics -␣
↪Instrumentation and Methods for Astrophysics},
        year = 2021,
        month = jun,
    volume = {254},
    number = {2},
        eid = {22},
        pages = {22},
        doi = {10.3847/1538-4365/abef67},
archivePrefix = {arXiv},
    eprint = {2012.01426},
primaryClass = {astro-ph.GA},
    adsurl = {https://ui.adsabs.harvard.edu/abs/2021ApJS..254...22J},
    adsnote = {Provided by the SAO/NASA Astrophysics Data System}
}
```

CHAPTER

NINETEEN

CHANGELOG

19.1 v1.2.0 (2022-12-31)

- Document, improvements, and bugfixes in `LineSpecModel` (h/t @kgarofali)
- Add `AGNSpecModel` with a scalable, empirical AGN emission line template.
- Fix floating point issue with Dirichlet SFH transforms.
- Implement `nested_target_n_effective` as dynesty stopping criterion.
- Fixes to the dynesty interface for dynesty >= 2.0 (h/t @mjastro)
- Fix sign error in Powell minimization (h/t @blanton144)
- Fix bugs in parameter template for emission line fitting.
- numeropus documentation updates including nebular emission details.

19.2 v1.1.0 (2022-02-20)

- Improved treatment of emission lines in `SpecModel`, including ability to ignore selected lines entirely.
- New `NoiseModelKDE` and `Kernel` classes to accommodate non-Gaussian and correlated uncertainties, courtesy of @wpb-astro
- New flexible SFH parameterization courtesy @wrensuess
- Support for `sedpy.observate.FilterSet` objects and computing rest-frame absolute magnitudes.
- Documentation updates, including a dedicated SFH page and a quickstart.
- Several bugfixes including fixes to the "logm_sfh" parameter template, a fix for the nested sampling argument parsing, and bestfit spectrum saving.

19.3 v1.0 (2021-12-02)

Release to accompany submitted paper. Includes

- New plotting module
- Demonstrations of MPI usage with dynesty
- Numerous small bugfixes.

19.4 v0.4 (2021-07-08)

- New `models.SpecModel` class that handles much of the conversion from FSPS spectra to observed frame spectra (redshifting, smoothing, dimming, spectroscopic calibration, filter projections) internally instead of relying on source classes.
- The `SpecModel` class enables analytic marginalization of emission line amplitudes, with or without FSPS based priors.
- A new mixture model option in the likelihood to handle outlier points (for diagonal covariance matrices)
- A noise model kernel for photometric calibration offsets.
- Rename `mean_model()` to `predict()` (old method kept for backwards compatibility)
- Some fixes to priors and optimization
- Python3 compatibility improvements (now developed and tested with Python3)

19.5 v0.3 (2019-04-23)

- New UI, based on `argparse` command line options and a high level ``fit_model()` function that can use emcee, dynesty, or optimization algorithms
- New `prospector_parse` module that generates a default argument parser.
- Importable default probability function as `fitting.lnprobfn()`
- Non-object prior methods removed
- Documentation and new notebook reflect UI changes
- `model_setup` methods are deprecated, better usage of warnings

PYTHON MODULE INDEX

p

INDEX

T

U

Z